Humboldt University

Computer Science Department
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Operating Systems Principles

Lab 1 – Linux Kernel

Das Praktikum

- Alle zwei Woche eine Übung
  - **Termin 1: Donnerstag**
    - 11:15 – 12:45 Uhr
    - RUD 26, Raum 1‘305
  - **Termin 2: Freitag**
    - 11:15 – 12:45 Uhr
    - RUD 26, Raum 1‘305

- Arbeitsgruppen zu ~2-3 Personen (max. 25 Gruppen!)

- Erfolgreiches Praktikum ist Voraussetzung für Prüfungszulassung

Systems Architecture Group
http://sar.informatik.hu-berlin.de

Das Praktikum

- **3 - 4 Aufgaben (geplant)**
  - Aufgabenstellung auf der SAR Website
  - ca. 2 Wochen Zeit
  - Abgabe über GOYA

- Praktikumsablauf
  - Erläutern der aktuellen Aufgabe
  - Besprechen der vorherigen Aufgabe (erst nach Abgabe !)
  - Evtl. kleine Übungen/Fragen zum aktuellen Thema der Vorlesung
  - Fragen von Euch

Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

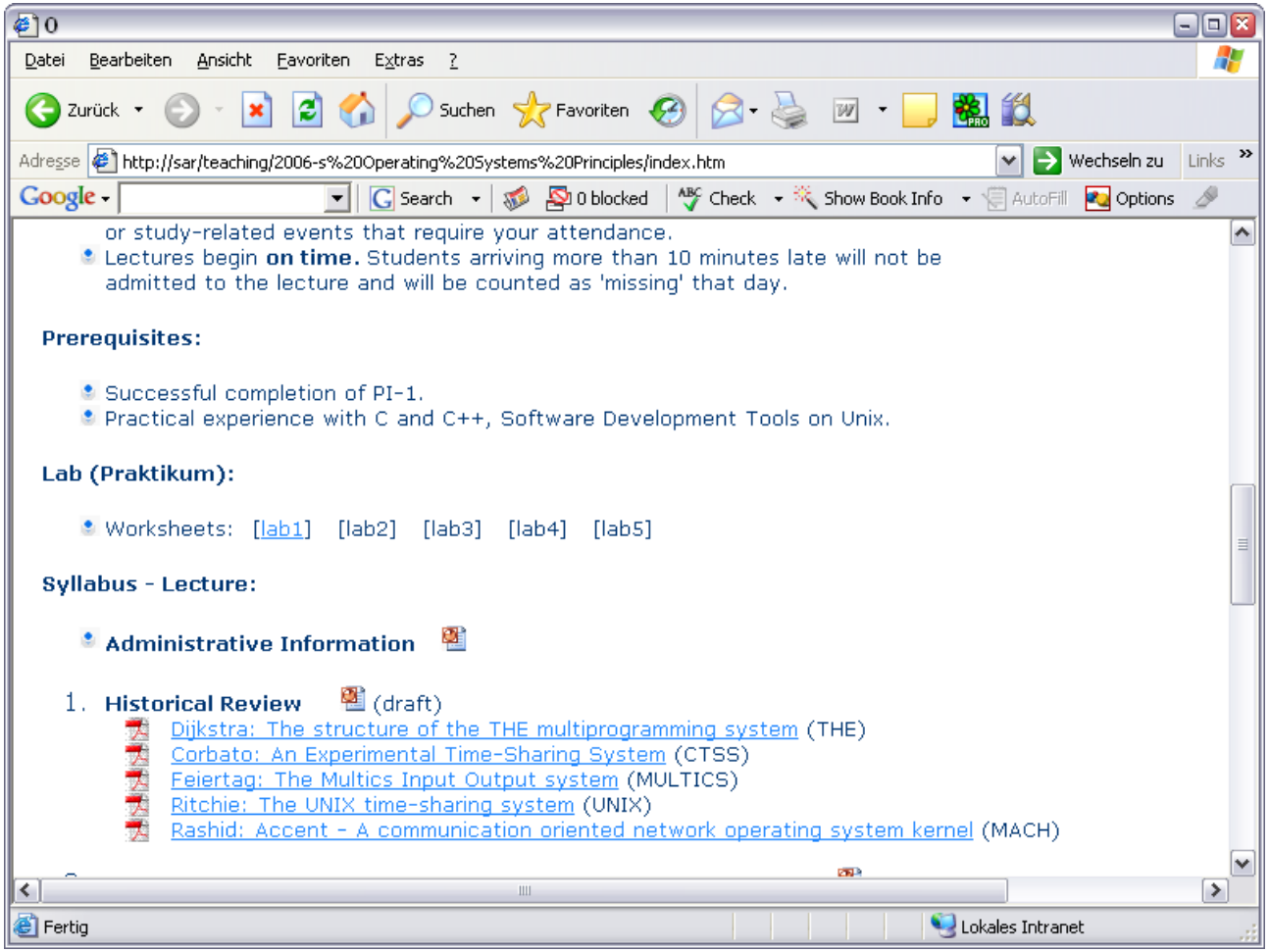
Organisatorisches

Datum	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
16.04. - 20.04.				**Praktikum 1**	**Erste Aufgabe (Beginn)** **Praktikum 3**
23.04. – 27.04.				**Praktikum 2**	**Praktikum 4**
30.04. – 04.05.				**Tag der Informatik**	**Zweite Aufgabe (Beginn)** **Praktikum 3+1**
07.05. – 11.05.	**Erste Aufgabe (Abgabe)**			**Praktikum 2+1**	**Praktikum 4+1**
14.05. – 18.05.				**Frei (Himmelfahrt)**	**Dritte Aufgabe (Beginn)** **Praktikum 3 + 1**
21.05. – 25.05.	**Zweite Aufgabe (Abgabe)**			**Praktikum 2 + 1**	**Praktikum 4 + 1**
28.05. – 01.06.				**Praktikum 1**	**Vierte Aufgabe (Beginn)** **Praktikum 3**
04.06. – 08.06.	**Dritte Aufgabe (Abgabe)**			**Praktikum 2**	**Praktikum 4**
11.06. – 15.06.				**Praktikum 1**	**Praktikum 3**
18.06. – 22.06.	**Vierte Aufgabe (Abgabe)**			**Praktikum 2**	**Praktikum 4**
25.06. – 29.06.				**Praktikum 1**	**Praktikum 3**
02.07. – 06.07.				**Praktikum 2**	**Praktikum 4**
09.07. – 13.07.					

Erstes Praktikum

- 2 Wochen Zeit zum Lösen der Aufgaben
- In dieser Woche
  - Erläutern der Aufgabenstellung
  - Erläuterung zur VM
- Nächste Woche
  - Zeit zur Bearbeitung
- Übernächste Woche
  - Zeit zur Bearbeitung
  - Konsultation, wenn gewünscht
- …
- Nächstes Treffen: **03/04.05.2012**

Praktikumsaufgaben



- Neue Aufgaben: jeweils Freitag
- Abgabe: Montag, 2 Wochen später

Systems Architecture Group
http://sar.informatik.hu-berlin.de

Praktikum

Requirements:

- Practical experience with C and C++, Software Development Tools on Unix.

- Virtual Machines (Qemu)
  - Download image via scp (alex.inform….)
  - You will be root / superuser
    - You crash the machine → YOU fix it!
    - You crash the network → YOU …

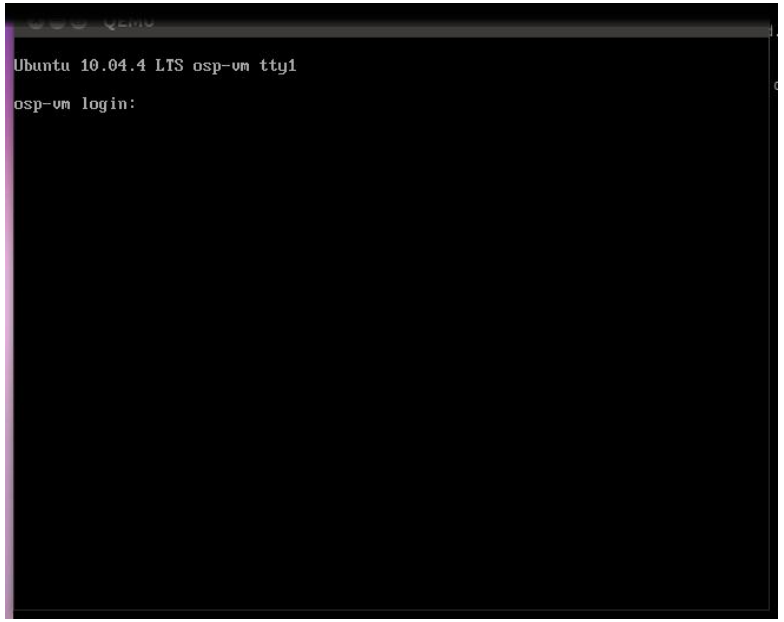
Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Virtual Machines

- Qemu Virtual Machine
  - RAM 256MB
  - Hard disk 1GB
  - Access to the Internet
  - Running Linux (Ubuntu 10.04.4 LTS)
  - User & Root Access
  - sshfs (Remote filesystem, e.g. home directory)
  - Second hard disk (/osp)
- Qemu is installed on Linux PC (Linux Pool)

Virtual Machines

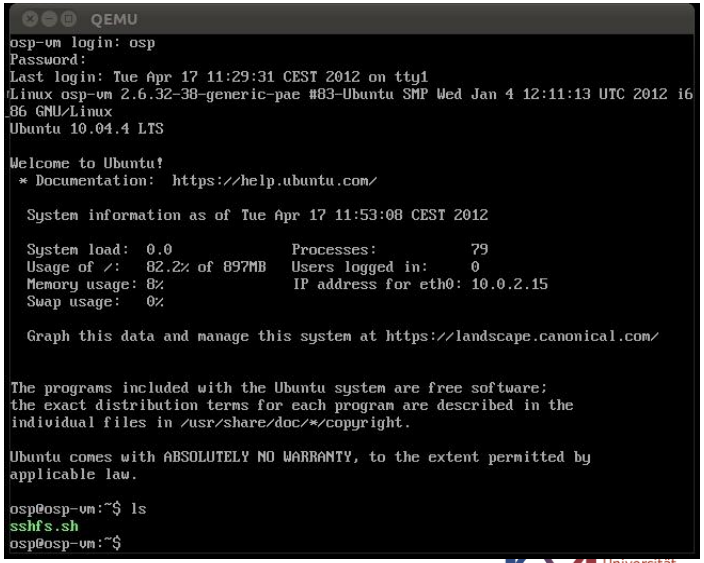
- Installation
  - Install Qemu:
    - Ubuntu: apt-get install qemu
    - Windows: download qemu-0.13.0-windows.zip
  - Download images and scripts:
    - scp –r user@alex.informatik.hu-berlin.de:/home/OSP1/osp_image .
    - Windows: copy files to qemu-directory
- Start qemu:
  - Linux: ./osp_start.sh
  - Windows: osp_start.bat



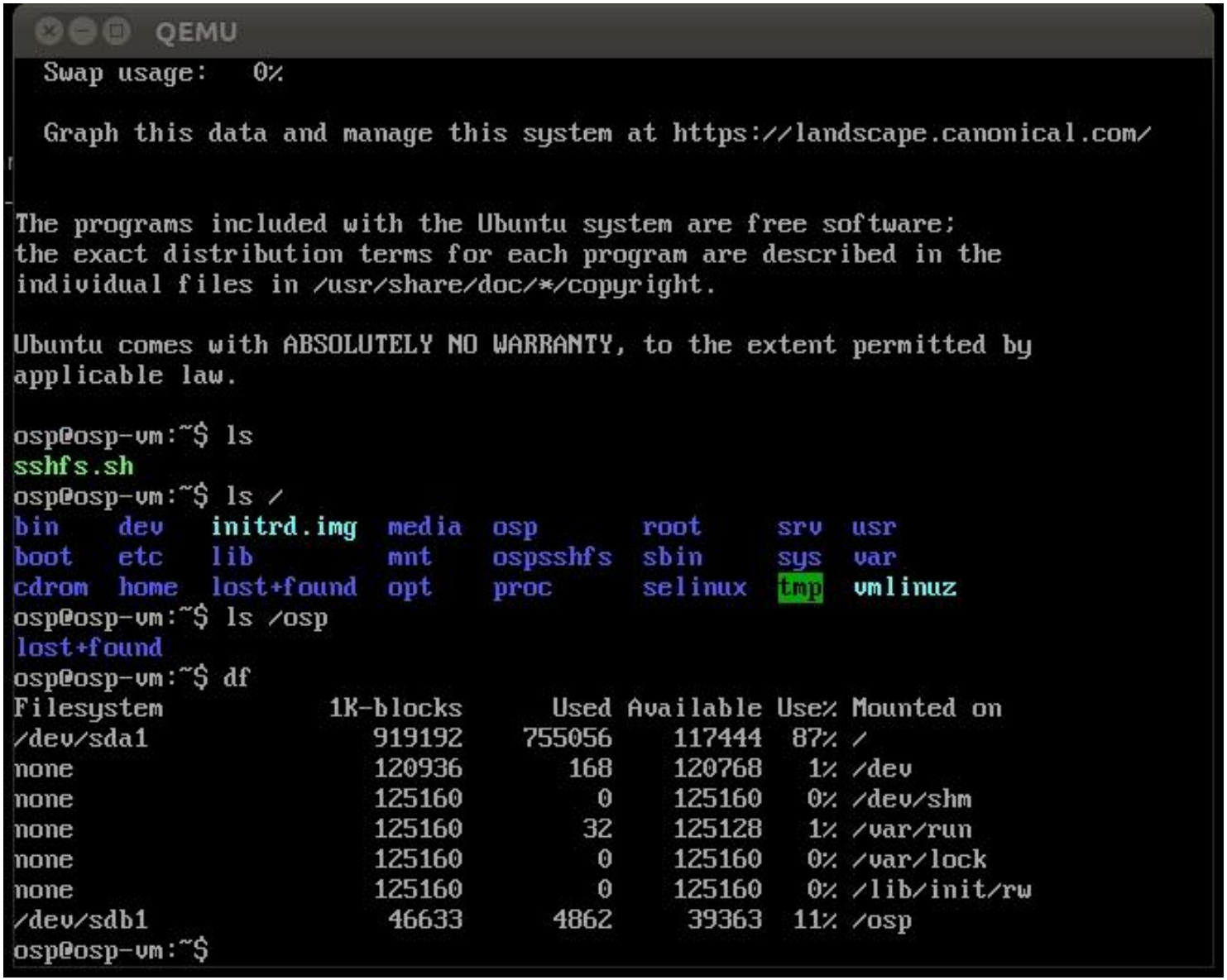
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Virtual Machines: Tools

- Login:
  - Username: osp
  - Passwd: osp
  - Need root shell?: sudo /bin/bash + passwd: „osp“
- sshfs.sh:
  - ./sshfs.sh username : mount home directory (gruenau.inf….)
  - ./sshfs.sh username host : mount home dir of user:
    - sshd must be installed on host !!!
  - ./sshfs.sh : umount /ospsshfs
  - Home directory is mounted on /ospsshfs
- Second hard drive:
  - Mounted on /osp
  - 50 MB



Virtual Machines: Tools



Systems Architecture Group
http://sar.informatik.hu-berlin.de

Virtual Machines

Questions??

Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Lab 1 explained

- Create a Linux kernel module: sarlkm.ko
- Create a pseudo-file /proc/sarlkm using a kernel module
  - Content:

    <prompt><working group>, <number of seconds since booting>
  - Test:
    - cat /proc/sarlkm
    - echo –n „newprompt“ > /proc/sarlkm
- Implement a new system call (kernel.prompt): get/set prompt of pseudo-file
  - Test tool: sysctl (man sysctl)
- Add module parameter (prompt_param) to set the prompt
- Write a command line program to get/set the hostname via the system call
- Additional exercises: Compile and install the Linux Kernel

Loadable Kernel Modules

- A great mechanism for OS ‘extensibility’
- Kernel can be modified while it’s running
- No need to recompile and then reboot

- But inherently unsafe: any ‘bug’ can cause a system malfunction or a complete crash!
  - Only authorized ‘system administrators’ are allowed to install kernel modules

‘insmod’ and ‘rmmod’

- root is allowed to ‘install’ kernel objects:

```
$  /sbin/insmod  myLKM.ko
```

- root is allowed to ‘remove’ kernel objects:

```
$  /sbin/rmmod  myLKM
```

- Anyone is allowed to ‘list’ kernel objects:

```
$  /sbin/lsmod
```

Creating a new LKM

- You can use any text-editor (e.g., ‘vi’) to create the source-code (in C) for a Linux kernel module (e.g., mod.c)

- But a kernel module differs from a normal C application program
  - No libc (standard C runtime)
  - No C++ and related libraries
  - No memory protection
  - No floating point support
  - Small, fixed-size stack

- For any LKM, two entry-points are mandatory
  - `init_module()`
  - `cleanup_module()`

Linux module structure

- Two ‘module administration’ functions are mandatory components in every module

plus

- Appropriate ‘module service’ functions and their supporting kernel data-structures are optional components in particular modules

also

- Recent kernels require a Module License!

Required module functions

```
int  init_module( void );
```

// gets called during module installation

```
void  cleanup_module( void );
```

// gets called during module removal

Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

A minimal module-template

```
#include <linux/module.h>

int init_module( void )
{
        // code here gets called during module installation
}

void cleanup_module( void )
{
        // code here gets called during module removal
}

MODULE_LICENSE("GPL");
```


Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

How to compile a module

- Use the ‘make’ utility (Makefile):

```
KERNEL_VERSION      := `uname -r`
KERNEL_DIR          := /lib/modules/$(KERNEL_VERSION)/build
INSTALL_MOD_DIR     := .
PWD                 := $(shell pwd)

obj-m               := hellolkm.o
hellolkm-objs        := hello.o

all: hellolkm

hellolkm:
    @echo "Building module...“
    @(cd $(KERNEL_DIR) && make -C $(KERNEL_DIR) SUBDIRS=$(PWD) CROSS_COMPILE=$(CROSS_COMPILE)
    modules)
```

The ‘printk()’ function

- Kernel modules cannot call any functions in the C runtime library, e.g., ‘printf()’
- But similar kernel versions of important functions are provided, e.g., ‘`printk()`’
- Syntax and semantics are slightly different
  - e.g., priority and message-destination
- Capabilities may be somewhat restricted
  - e.g., printk() can’t show floating-point

Systems Architecture Group
http://sar.informatik.hu-berlin.de

Simple module example

```
#include <linux/module.h>

int init_module( void )
{
        printk( "<1>Hello, world!\n" );
        return  0;    // SUCCESS
}

void cleanup_module( void )
{
        printk( "<1>Goodbye everyone\n" );
}
MODULE_LICENSE("GPL");
```


Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

How to install and remove

```
#   /sbin/insmod   hello.ko

#   /sbin/rmmod   hello
```


Systems Architecture Group
http://sar.informatik.hu-berlin.de

Creating proc files

- Let's see how we can use kernel functions to create our own **`/proc`** file
- Easy if we use **`create_proc_entry()`** during module-initialization (and **`remove_proc_entry()`** during cleanup)

```
static inline struct proc_dir_entry*
  create_proc_entry(
    const char *name,
    mode_t mode,
    struct proc_dir_entry *base)
```

- Set read/write for procfile using proc_dir_entry - member:
  - read_proc & write_proc

Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Creating proc files

- Example

```
struct proc_dir_entry* my_proc_entry;
my_proc_entry =
  create_proc_entry("myproc", S_IRUGO, NULL);

my_proc_entry->read_proc = procfile_read;
my_proc_entry->write_proc = procfile_write;
```

- More info:
  - http://www.tldp.org/LDP/lkmpg/2.6/html/x769.html

jiffies

- `unsigned long volatile jiffies;`

- global kernel variable (used by scheduler)
- Initialized to zero when system reboots
- Gets incremented when timer interrupts
- Counts ‘clock-ticks’ since CPU restart
- ‘tick-frequency’ is architecture dependent

Writing the ‘jiffies.c’ module

- We need to declare a name for pseudo-file

```
static char modname[ ] = “jiffies”;
```

- We need to define a proc_read() function
  (see code on the following slide)

- We need to ‘register’ our filename, along with its ‘read’ method, in init_module()

- We need to ‘unregister’ our pseudo-file in cleanup_module()

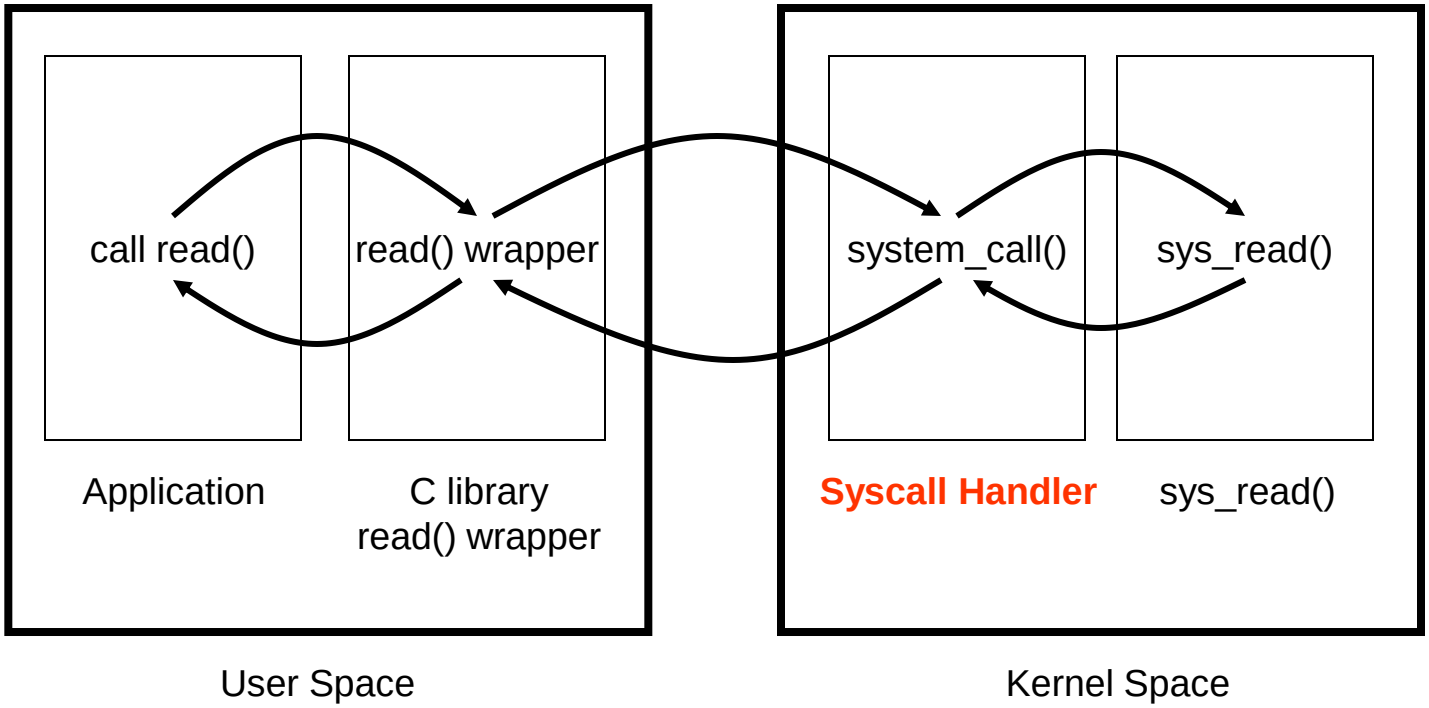
Writing the ‘jiffies.c’ module

```
static int  my_proc_read( char *buf,  char **start,
     off_t off,  int count,  int *eof,  void *data )
{
     return sprintf( buf, "jiffies=%lu\n", jiffies );
}

int init_module( void )
{
        struct proc_dir_entry *my_proc_entry;
        my_proc_entry = create_proc_entry( modname, S_IRUGO, NULL );
         my_proc_entry->read_proc = my_proc_read;
        return  0;
}

void cleanup_module( void )
{
      remove_proc_entry( modname, NULL );
}
```

System Calls

- ***System Call*** – the method used by a process to request action by the operating system.
  - Usually takes the form of a *trap* to a specific location in the interrupt vector.
  - Control passes through the *interrupt vector to a service routine* in the OS, and the mode bit is set to monitor mode.
  - The monitor verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call.
- Three general methods are used to pass parameters between a running program and the operating system.
  - Pass parameters in *registers*.
  - Store the parameters in a *table in memory*, and pass the *table address* in a register.
  - *Push* parameters onto the *stack* by the program, and *pop* off the stack by operating system.

System Call Handler



Linux System Calls

- User program invokes system calls to access kernel services
  - e.g. getpid()

- In User Space, system calls are implemented as wrapper functions (libc).
- System calls are defined using macros:

  `_syscall0()` ... **`_syscall5()`** … see include/asm-i386/unistd.h

```
pid_t getpid()
   _syscall0(int,getpid)
unsigned int alarm(unsigned int seconds)
   _syscall1(unsigned int,alarm,unsigned int,seconds)
int write(int fd, const char * buf, unsigned int count)
   _syscall3(int,write,int,fd,const char *,buf,unsigned int,count)
```

- If failed: return -1 and set errno

Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Transfer of Control

- By special programmed exception (**int $0x80**)
  - ”int <NUMBER>”: instruction to trigger an interrupt (number)
- Exception vector 128 (0x80) defined as “system call”
- Inside the kernel:
  - System call handler **`system_call()`** written in assembly code
  - Dispatches to the corresponding ***system call service routine***, e.g., **sys_alarm(),** according to system call number **__NR_alarm**
  - System call dispatch table (see arch/i386/kernel/entry.S)

```
.data
ENTRY(sys_call_table)
.long SYMBOL_NAME(sys_ni_syscall) /* 0 */
.long SYMBOL_NAME(sys_exit)
.long SYMBOL_NAME(sys_fork)
.long SYMBOL_NAME(sys_read)
.long SYMBOL_NAME(sys_write)
.long SYMBOL_NAME(sys_open) /* 5 */
.long SYMBOL_NAME(sys_close)
```

System Call Service Routine

Corresponding to each system call, e.g.:

```
asmlinkage unsigned long sys_alarm(unsigned int seconds)
{
    struct itimerval it_new, it_old;
    unsigned int oldalarm;

    it_new.it_interval.tv_sec = it_new.it_interval.tv_usec = 0;
    it_new.it_value.tv_sec = seconds;
    it_new.it_value.tv_usec = 0;
    do_setitimer(ITIMER_REAL, &it_new, &it_old);
    oldalarm = it_old.it_value.tv_sec;
    if (it_old.it_value.tv_usec)
        oldalarm++;
    return oldalarm;
}
```

Macro Expansion for alarm()

```
unsigned int alarm(unsigned int seconds)
{ long __res;
  __asm__ volatile ("int $0x80"
            : "=a" (__res)
            : "" (__NR_alarm), "b“ ((long)(seconds)));
  do {
      if ((unsigned long)(__res) >= (unsigned long)(-125)) {
          errno = -(__res);
          __res = -1;
      }
      return (unsigned int) (__res);
  } while (0);
}
```


Systems Architecture Group
http://sar.informatik.hu-berlin.de

Parameter Passing

- Through CPU Registers
  - Each parameter is a word (32bit on 32bit architectures)
  - Parameters limited by register number

- What about complex data (e.g., buffer, such as in write())?
  - Parameter: memory address
  - Kernel will copy data from/to process' user-mode memory space

- Two service routines

```
copy_from_user(void*to,const void *from,unsigned long n );
copy_to_user(void*to, const void *from, unsigned long n );
```

Adding New System Calls

```
struct ctl_table_header * register_sysctl_table(ctl_table *
table, int insert_at_head);

void unregister_sysctl_table(struct ctl_table_header *
table);
```

- Fields of `ctl_table *`:
  - int ctl_name : a numeric id, unique in each table
  - const char *procname : corresponding name in /proc
  - void *data : pointer to data, e.g. point to an integer value for integer items
  - int maxlen : size of pointed data
  - mode_t mode : octal mode of the file
  - ctl_table *child : the child table for dirs (NULL for leaf nodes
  - proc_handler *proc_handler : handler to perf any read/write spawned by /proc files
  - ctl_handler *strategy : handler for reads/writes using system calls
  - …
- More infos: http://www.linux.it/~rubini/docs/sysctl/sysctl.html

Why Not to Implement a System Call

- The pros of implementing a new interface as a syscall
  - System calls are simple to implement and easy to use
  - System calls performance on Linux is blindingly fast.
- The cons
  - You need a syscall number, which needs to be officially assigned to you during a developmental kernel series.
  - After the system call is in a stable series kernel, it is written in stone.
    - The interface cannot change without breaking user-space applications
  - Each architecture needs to separately register the system call and support it.
  - For simple exchanges of information, a system call is overkill
- Alternatives
  - Use the file interface (read, write, ioctl, …)
  - Procfs or sysfs

Using syscall – userspace application

```
int sysctl (int *name, int nlen,
            void *oldval, size_t *oldlenp,
            void *newval, size_t newlen);
```

- Parameter: **name**
  - **Array of integer (sysctl item: dir or leaf node)**
    - **See <linux/sysctl.h)**
- Parameter: **nlen**
  - Length of array „name“
- Parameter: **oldval/oldlenp**
  - Pointer to data buffer for old value
- Parameter: **newval/newlen**
  - Pointer to data buffer for new value
  - Read: newval = NULL
- More infos: man sysctl (manpages)

Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Add module parameter

```
char *my_param = „foobar“; //default
module_param(my_param, charp, 0600);
MODULE_PARM_DESC(my_param, „Set my param“);
```

- **module_param:**
  - 1. parameter: name of the variable
  - 2. parameter: type (int, charp,…)
  - 3. permissions for corresponding file (sysfs)

- Set param: insmod myLKM.ko my_param=„value“
- Info about module: modinfo myLKM

- More infos:
  - http://www.tldp.org/LDP/lkmpg/2.6/html/

Systems Architecture Group
http://sar.informatik.hu-berlin.de